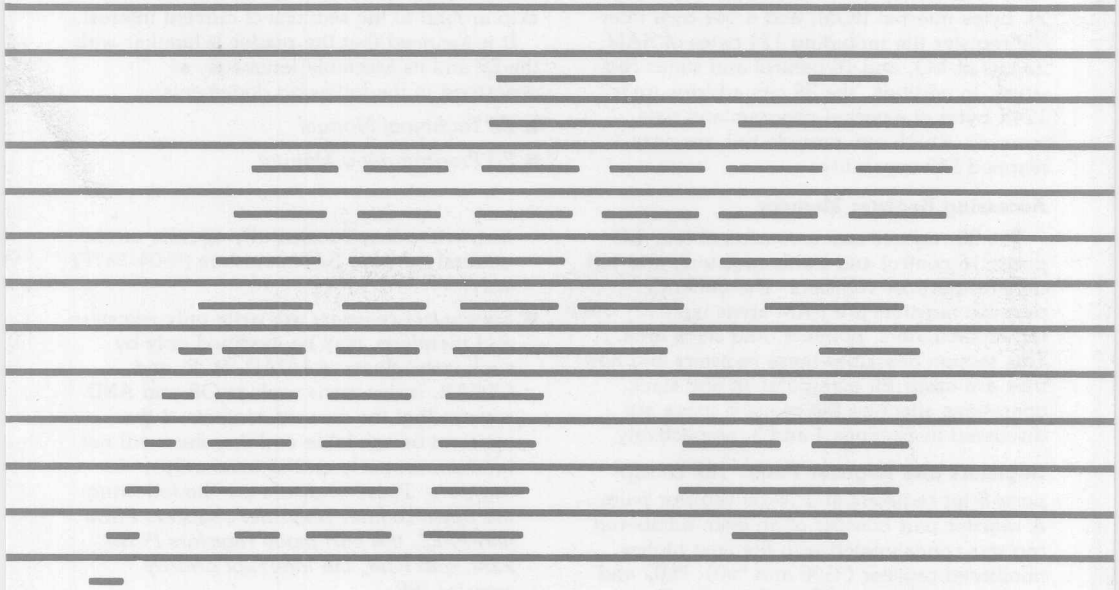
SGS μProduct Support 5



A Programmer's Guide to the Z8 Microcomputer

Introduction

The Z8 is the first microcomputer to offer both a highly integrated microcomputer on a single chip and a fully expandable microprocessor for I/O-and memory-intensive applications. The Z8 has two timer/counters, a UART, 2K bytes internal ROM, and a 144-byte internal register file including 124 bytes of RAM, 32 bits of I/O, and 16 control and status registers. In addition, the Z8 can address up to 124K bytes of external program and data memory, which can provide full, memory-mapped I/O capability.

This application note describes the important features of the Z8, with software examples that illustrate its power and ease of use. It is divided into sections by topic; the reader need not read each section sequentially, but may skip around to the sections of current interest.

It is assumed that the reader is familiar with the Z8 and its assembly language, as described in the following documents:

- *Z8 Technical Manual*
- *Z8 Programming Manual*

Accessing Register Memory

The Z8 register space consists of four I/O ports, 16 control and status registers, and 124 general-purpose registers. The general-purpose registers are RAM areas typically used for accumulators, pointers, and stack area. This section describes these registers and how they are used. Bit manipulation and stack operations affecting the register space are discussed in Sections 4 and 5, respectively.

Registers and Register Pairs. The Z8 supports 8-bit registers and 16-bit register pairs. A register pair consists of an even-numbered register concatenated with the next higher numbered register (%00 and %01, %02 and %03, ... %7E and %7F, %F0 and %F1, ... %FE and %FF). A register pair must be addressed by reference to the even-numbered register. For example,

%F1 and %F2 is not a valid register pair;
%F0 and %F1 is a valid register pair, addressed by reference to %F0.

Register pairs may be incremented (INCW) and decremented (DECW) and are useful as pointers for accessing program and external data memory. Section 3 discusses the use of register pairs for this purpose.

Any instruction which can reference or modify an 8-bit register can do so to any of the 144 registers in the Z8, regardless of the inherent nature of that register. Thus, I/O ports, control, status, and general-purpose registers may all be accessed and manipulated without the need for special-purpose instructions. Similarly, instructions which reference or modify a 16-bit register pair can do so to any of the valid 72 register pairs. The only exceptions to this rule are:

- The DJNZ (decrement and jump if non-zero) instruction may successfully operate on the general-purpose RAM registers (%04-%7F) only.
- Six control registers are write-only registers and therefore, may be modified only by such instructions as LOAD, POP, and CLEAR. Instructions such as OR and AND require that the current contents of the operand be readable and therefore will not function properly on the write-only registers. These registers are the following: *the timer/counter prescaler registers PRE0 and PRE1, the port mode registers P01M, P2M, and P3M, the interrupt priority register IPR.*

Register Pointer. Within the register addressing modes provided by the Z8, a register may be specified by its full 8-bit address (0-%7F, %F0-%FF) or by a short 4-bit address. In the latter case, the register is viewed as one of 16 working registers within a working register group. Such a group must be aligned on a 16-byte boundary and is addressed by Register Pointer RP (%FD). As an example, assume the Register Pointer contains %70, thus pointing to the working register group from %70 to %7F. The LD instruction may be used to initialize register %76 to an immediate value in one of two ways:

```
LD  %76,#1   !8-bit register address is given
             by instruction (3 byte instruc-
             tion)!
   or
LD  R6,#1    !4-bit working register address
             is given by instruction; 4-bit
             working register group
             address is given by Register
             Pointer (2 byte instruction)!
```

Accessing Register Memory (Continued)

The address calculation for the latter case is illustrated in Figure 1. Notice that 4-bit working-register addressing offers code compactness and fast execution compared to its 8-bit counterpart.

To modify the contents of the Register Pointer, the Z8 provides the instruction

SRP #value

Execution of this instruction will load the upper four bits of the Register Pointer; the lower four bits are always set to zero. Although a load instruction such as

LD RP,#value

could be used to perform the same function, SRP provides execution speed (six vs. ten cycles) and code space (two vs. three bytes) advantages over the LD instruction. The instruction

SRP #%70

is used to set the Register Pointer for the above example.

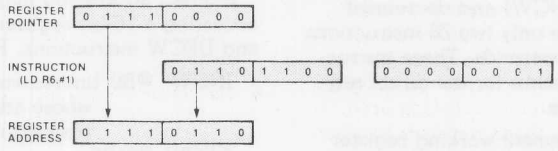


Figure 1. Address Calculation Using the Register Pointer

Context Switching. A typical function performed during an interrupt service routine is context switching. Context switching refers to the saving and subsequent restoring of the program counter, status, and registers of the interrupted task. During an interrupt machine cycle, the Z8 automatically saves the Program Counter and status flags on the stack. It is the responsibility of the interrupt service routine to preserve the register space. The recommended means to this end is to allocate a specific portion of the register file for use by the service routine. The service routine thus preserves the register space of the interrupted task by avoiding modification of registers not allocated as its own. The most efficient scheme with which to implement this function in the Z8 is to allocate a working register group (or portion thereof) to the interrupt service routine. In this way, the preservation of the interrupted task's registers is solely a matter of saving the Register Pointer on entry to the service routine, setting the Register Pointer to its own working register group, and restoring the Register Pointer prior to exiting the service routine. For example, assume such a register allocation scheme has been implemented in which the interrupt service routine for IRQ0 may access only working register Group 4 (registers %40–%4F). The service routine for IRQ0 should be headed by the code sequence:

```
PUSH RP    !preserve Register Pointer of
           interrupted task!
SRP  #%40  !address working register
           group 4!
```

Before exiting, the service routine should execute the instruction

POP RP

to restore the Register Pointer to its entry value.

It should be noted that the technique described above need not be restricted to interrupt service routines. Such a technique might prove efficient for use by a subroutine requiring intermediate registers to produce its outputs. In this way, the calling task can assume that its environment is intact upon return from the subroutine.

Addressing Mode. The Z8 provides three addressing modes for accessing the register space: Direct Register, Indirect Register, and Indexed.

Direct Register Addressing. This addressing mode is used when the target register address is known at assembly time. Both long (8-bit) register addressing and short

Accessing Register Memory (Continued)

(4-bit) working register addressing are supported in this mode. Most instructions supporting this mode provide access to single 8-bit registers. For example:

```
LD   %FE,#HI STACK
                !load register %FE (SPH) with
                 the upper 8-bits of the label
                 STACK!
AND 0,MASK_REG
                !AND register 0 with register
                 named MASK_REG!
OR   1,R5    !OR register 1 with working
                 register 5!
```

Increment word (INCW) and decrement word (DECW) are the only two Z8 instructions which access 16-bit operands. These instructions are illustrated below for the direct register addressing mode.

```
INCW  RR0  !increment working register
                 pair R0, R1:
                 R1 ← R1 + 1
                 R0 ← R0 + carry!
DECW  %7E
                !decrement working register
                 pair %7E, %7F:
                 %7F ← %7F – 1
                 %7E ← %7E – carry!
```

Note that the instruction

```
INCW  RR5
```

will be flagged as an error by the assembler (RR5 not even-numbered).

Indirect Register Addressing. In this addressing mode, the operand is pointed to by the register whose 8-bit register address or 4-bit working register address is given by the instruction. This mode is used when the target register address is not known at assembly time and must be calculated during program execution. For example, assume registers %60-%7F contain a buffer for output to the serial line via repetitive calls to procedure SERIAL_OUT. SERIAL_OUT expects working register 0 to hold the output character. The following instructions illustrate the use of the indirect addressing mode to accomplish this task:

```
LD     R1,#%20
                !working register 1 is the byte
                 counter: output %20 bytes!
LD     R2,#%60
                !working register 2 is the buf-
                 fer pointer register!
out_again:
LD     R0,@R2
                !load into working register 0
                 the byte pointed to by working
                 register 2!
INC    R2     !increment pointer!
CALL SERIAL_OUT
                !output the byte!
DJNZ   R1,out _again
                !loop till done!
```

Indirect addressing may also be used for accessing a 16-bit register pair via the INCW and DECW instructions. For example,

```
INCW  @R0 !increment the register pair
                 whose address is contained in
                 working register 0!
DECW  @%7F
                !decrement the register pair
                 whose address is contained in
                 register %7F!
```

The contents of registers R0 and %7F should be even numbers for proper access; when referencing a register pair, the least significant address bit is forced to the appropriate value by the Z8. However, the register used to point to the register pair need not be an even-numbered register.

Since the indirect addressing mode permits calculation of a target address prior to the desired register access, this mode may be used to simulate other, more complex addressing modes. For example, the instruction

```
SUB  4,BASE(R5)
```

requires the indexed addressing mode which is not directly supported by the Z8 SUBtract instruction. This instruction can be simulated as follows:

```
LD    R6,#BASE
                !working register 6 has the
                 base address!
ADD R6,R5  !calculate the target address!
SUB  4,@R6 !now use indirect addressing to
                 perform the actual subtract!
```

Any available register or working register may be used in place of R6 in the above example.

Accessing Register Memory (Continued)

Indexed Addressing. The indexed addressing mode is supported by the load instruction (LD) for the transference of bytes between a working register and another register. The effective address of the latter register is given by the instruction which is offset by the contents of a designated working (index) register. This addressing mode provides efficient memory usage when addressing consecutive bytes in a block of register memory, such as a table or a buffer. The working register used as the index in the effective address calculation can serve the additional role of counter for control of a loop's duration.

For example, assume an ASCII character buffer exists in register memory starting at address BUF for LENGTH bytes. In order to determine the logical length of the character string, the buffer should be scanned backward until the first nonoccurrence of a blank character. The following code sequence may be used to accomplish this task:

```
   LD    R0,#LENGTH
                 !length of buffer!
                 !starting at buffer end, look for
                  1st non-blank!
loop:
   LD    R1,BUF-1(R0)
   CP    R1,#' '
   JR    ne,found
                 !found non-blank!
   DJNZ  R0,loop
                 !look at next!
all_blanks:      !length = 0!
found:
```

5 instructions
12 bytes
1.5 μs overhead
10.5 μs (average) per character tested

At labels "all_blanks" and "found," R0 contains the length of the character string. These labels may refer to the same location, but they are shown separately for an application where special processing is required for a string of zero length. To perform this task without indexed addressing would require a code sequence such as:

```
   LD    R1,#BUF+LENGTH-1
   LD    R0,#LENGTH
                 !starting at buffer end, look for
                  1st non-blank!
loop1:
   CP    @R1,#' '
   JR    ne,found1
                 !found non-blank!
   DEC   R1      !dec pointer!
   DJNZ  R0,loop1
                 !are we done?!
all_blanks1:     !length = 0!
found1:
```

6 instructions
13 bytes
3 μs overhead
9.5 μs (average) per character tested

The latter method requires one more byte of program memory than the former, but is faster by four execution cycles (1 μs) per character tested.

As an alternate example, assume a buffer exists as described above, but it is desired to scan this buffer forward for the first occurrence of an ASCII carriage return. The following illustrates the code to do this:

```
   LD    R0,#-LENGTH
                 !starting at buffer start, look for
                  1st carriage return (=%0D)!
next:
   LD    r1,BUF+LENGTH(R0)
   CP    R1,#%0D
   JR    eq,cr   !found it!
   INC   R0      !update counter/index!
   JR    nz,next
                 !try again!
cr:
   ADD   R0,#LENGTH
                 !R0 has length to CR!
```

7 instructions
16 bytes
1.5 μs overhead
12 μs (average) per character tested

Accessing Program and External Data Memory

In a single instruction, the Z8 can transfer a byte between register memory and either program or external data memory. Load Constant (LDC) and Load Constant and Increment (LDCI) reference program memory; Load External (LDE) and Load External and Increment (LDEI) reference external data memory. These instructions require that a working register pair contain the address of the byte in either program or external data memory to be accessed by the instruction (indirect working register pair addressing mode). The register byte operand is specified by using the direct working register addressing mode in LDC and LDE or the indirect working register addressing mode in LDCI and LDEI. In addition to performing the designated byte transfer, LDCI and LDEI automatically increment both the indirect registers specified by the instruction. These instructions are therefore efficient for performing block moves between register and either program or external data memory. Since the indirect addressing mode is used to specify the operand address within program or external data memory, more complex addressing modes may be simulated as discussed earlier in Section 2.4.2. For example, the instruction

```
LDC   R3,BASE(R2)
```

requires the indexed addressing mode, where BASE is the base address of a table in program memory and R2 contains the offset from table start to the desired table entry. The following code sequence simulates this instruction with the use of two additional registers (R0 and R1 in this example).

```
LD    R0,#HI BASE
LD    R1,#LO BASE
            !RR0 has table start address!
ADD   R1,R2
ADC   R0,#0
            !RR0 has table entry address!
LDC   R3,@RR0
            !R3 has the table entry!
```

Configuring the Z8 for I/O Applications vs. Memory Intensive Applications. The Z8 offers a high degree of flexibility in memory and I/O intensive applications. Thirty-two port bits are provided of which 16, 12, eight, or zero may be configured as address bits to external memory. This allows for addressing of 62K, 4K or 256 bytes of external memory, which can be expanded to 124K, 8K, or 512 bytes if the Data Memory Select output ($\overline{DM}$) is used to distinguish between program and data memory accesses. The following instructions illustrate the code sequence required to configure the Z8 with 12 external addressing lines and to enable the Data Memory Select output.

```
LD    P01M,#%(2)00010010
            !bit 3-4: enable AD0-AD7;
             bit 0-1: enable A8-A11!
LD    P3M,#%(2)00001000
            !bit 3-4: enable DM!
```

The two bytes following the mode selection of ports 0 and 1 should not reference external memory due to pipelining of instructions within the Z8. Note that the load instruction to P3M satisfies this requirement (providing that it resides within the internal 2K bytes of memory).

LDC and LDE. To illustrate the use of the Load Constant (LDC) and Load External (LDE) instructions, assume there exists a hardware configuration with external memory and Data Memory Select enabled. The following module illustrates a program for tokenizing an ASCII input buffer. The program assumes there is a list of delimiters (space, comma, tab, etc.) in program memory at address DELIM for COUNT bytes (accessed via LDC) and that an ASCII input buffer exists in external data memory (accessed via LDE). The program scans the input buffer from the current location and returns the start address of the next token (i.e. the address of the first nondelimiter found) and the length of that token (number of characters from token start to next delimiter).

Accessing Program and External Data Memory (Continued)

```
Z8ASM     2.0
LOC    OBJ CODE   STMT SOURCE STATEMENT

                    1 SCAN    MODULE
                    2 CONSTANT
                    3  COUNT  :=       6
                    4 GLOBAL
                    5         $SECTION PROGRAM
P 0000 20  3B  2C   6 DELIM   ARRAY   [COUNT BYTE]    :=
P 0003 2E  0A  0D
                    7                  [' ' , ';' , ',' , '.' , %0A , %0D]
                    8
P 0006              9 scan    PROCEDURE
                   10 !****************************************************
                   11  Purpose =       To find the next token within an
                   12                   ASCII buffer.
                   13
                   14  Input =         RR0 = address of current location
                   15                       within input buffer in external
                   16                       memory.
                   17
                   18  Output =        RR4 = address of start of next token
                   19                   RR0 = address of new token's ending
                   20                         delimiter
                   21                   R2  = length of token
                   22                   R3  = ending delimiter
                   23                   R6,R7,R8,R9 destroyed
                   24
                   25 ****************************************************!
                   26 ENTRY
P 0006 B0  E2      27         clr     R2       !init. length counter!
                   28        DO
P 0008 82  30      29         LDE     R3,@RR0  !get byte from input buffer!
P 000A A0  E0      30         incw    RR0      !increment pointer!
P 000C D6  002E'   31         call    check    !look for non-delimiter!
P 000F FD  0015'   32         IF  C THEN
P 0012 8D  0018'   33           EXIT           !found token start!
                   34         FI
P 0015 8D  0008'   35        OD
                   36
P 0018 48  E0      37         ld      R4,R0
P 001A 58  E1      38         ld      R5,R1    !RR4 = token starting addr!
                   39        DO
P 001C 2E          40         inc     R2       !inc. length counter!
P 001D 82  30      41         LDE     R3,@RR0  !get next input byte!
P 001F D6  002E'   42         call    check    !look for delimiter!
P 0022 7D  0028'   43         IF  NC  THEN
P 0025 8D  002D'   44           EXIT           !found token end!
                   45         FI
P 0028 A0  E0      46         incw    RR0      !point to next byte!
P 002A 8D  001C'   47        OD
                   48
P 002D AF          49         ret
P 002E             50 END     scan
                   51
P 002E             52 check   PROCEDURE
                   53 !****************************************************
                   54  Purpose =       compare current character with
                   55                  delimiter table until table
                   56                  end or match found
                   57
                   58  input =         DELIM = start address of table
                   59                  COUNT = length of that table
                   60                  R3 = byte to be scrutinized
                   61
                   62  output =        Carry flag = 1 => input byte
                   63                  is not a delimiter (no match found)
```

Accessing Program and External Data Memory (Continued)

```
                        64
                        65                  Carry flag = 0 => input byte
                        66                  is a delimiter (match found)
                        67                  R6,R7,R8,R9  destroyed
                        68
                        69 ************************************************!
                        70 ENTRY
P 002E 6C  00*          71          ld      R6,#HI DELIM
P 0030 7C  00*          72          ld      R7,#LO DELIM    !RR6 points to
                        73                                   delimiter list!
P 0032 8C  06           74          ld      R8,#COUNT       !R8 = length of list!
                        75 here:
P 0034 C2  96           76          LDC     R9,@RR6         !get table entry!
P 0036 A0  E6           77          incw    RR6             !point to next entry!
P 0038 A2  93           78          cp      R9,R3           !R3 = delimiter?!
P 003A 6B  03           79          jr      eq,bye          !yes. carry = 0!
P 003C 8A  F6           80          djnz    R8,here         !next entry!
P 003E DF               81          scf                     !table done. R3
                        82                                   not a delimiter!
                        83 bye:
P 003F AF               84          ret
P 0040                  85 END      check
                        86 END      SCAN

   0 ERRORS
ASSEMBLY COMPLETE
```

27 instructions
58 bytes
Execution time is a function of the number of leading delimiters before token start (x) and the number of characters in the token (y): 123 µs overhead + 59x µs + 102y µs (average) per token

LDCI. A common function performed in Z8 applications is the initialization of the register space. The most obvious approach to this function is the coding of a sequence of "load register with immediate value" instructions (each occupying three program bytes for a register or two program bytes for a working register). This approach is also the most efficient technique for initializing less than eight consecutive registers or 14 consecutive working registers. For a larger register block, the LDCI instruction provides an economical means of initializing consecutive registers from an initialization table in program memory. The following code excerpt illustrates this technique of initializing control registers %F2 through %FF from a 14-byte array (INIT_tab) in program memory:

```
   SRP   #%00
                !RP not %F0!
   LD    R6,#HI INIT_tab
   LD    R7,#LO INIT_tab
   LD    R8,#%F2
                !1st reg to be initialized!
   LD    R9,#14
                !length of register block!
loop:
   LDCI  @R8,@RR6
                !load a register from the
                init table!
   DJNZ  R9,loop
                !continue till done!
```

7 instructions
14 bytes
7.5 µs overhead
7.5 µs per register initialized

Accessing Program and External Data Memory (Continued)

LDEI. The LDEI instruction is useful for moving blocks of data between external and register memory since auto-increment is performed on both indirect registers designated by the instruction. The following code excerpt illustrates a register buffer being saved at address %40 through %60 into external memory at address SAVE:

```
LD    R10,#HI SAVE
            !external memory!
LD    R11,#LO SAVE
            !address!
LD    R8,#%40
            !starting register!
LD    R9,#%21
            !number of registers to save in
             external data memory!
loop:
LDEI  @RR10,@R8
            !init a register!
DJNZ  R9,loop
            !until done!
```

6 instructions
12 bytes
6 μs overhead
7.5 μs per register saved

Bit Manipulations

Support ot the test and modification of an individual bit or group of bits is required by most software applications suited to the Z8 microcomputer. Initializing and modifying the Z8 control registers, polling interrupt requests, manipulating port bits for control of or communication with attached devices, and manipulation of software flags for internal control purposes are all examples of the heavy use of bit manipulation functions. These examples illustrate the need for such functions in all areas of the Z8 register space. These functions are supported in the Z8 primarily by six instructions:

- Test under Mask (TM)
- Test Complement under Mask (TCM)
- AND
- OR
- XOR
- Complement (COM)

These instructions may access any Z8 register, regardless of its inherent type (control, I/O, or general purpose), with the exception of the six write-only control registers (PRE0, PRE1, P01M, P2M, P3M, IPR) mentioned earlier in Section 2.1. Table 1 summarizes the function performed on the destination byte by each of the above instructions. All of these instructions, with the exception of COM, require a mask operand. The "selected" bits referenced in Table 1 are those bits in the destination operand for which the corresponding mask bit is a logic 1.

Opcode	Use
TM	To test selected bits for logic 0
TCM	To test selected bits for logic 1
AND	To reset all but selected bits to logic 0
OR	To set selected bits to logic 1
XOR	To complement selected bits
COM	To complement all bits

Table 1. Bit Manipulation Instruction Usage

The instructions AND, OR, XOR, and COM have functions common to today's microprocessors and therefore are not described in depth here. However, examples of the use of these instructions are laced throughout the remainder of this document, thus giving an integrated view of their uses in common functions. Since they are unique to the Z8, the functions of Test under Mask and Test Complement under Mask, are discussed in more detail next.

Test under Mask (TM). The Test under Mask instruction is used to test selected bits for logic 0. The logical operation performed is

destination AND source

Neither source nor destination operand is modified; the FLAGS control register is the only register affected by this instruction. The zero flag (Z) is set if all selected bits are logic 0; it is reset otherwise. Thus, if the selected destination bits are either all logic 1 or a combination of 1s and 0s, the zero flag would be cleared by this instruction. The sign flag (S) is either set or reset to reflect the result of the

Bit Manipulation (Continued)

AND operation; the overflow flag (V) is always reset. All other flags are unaffected. Table 2 illustrates the flag settings which result from the TM instruction on a variety of source and destination operand combinations. Note that a given TM instruction will never result in both the Z and S flags being set.

Test Complement under Mask. The Test Complement under Mask instruction is used to test selected bits for logic 1. The logical operation performed is

(NOT destination) AND source.

Destination	Source	Flags		
(binary)	(binary)	Z	S	V
10001100	01110000	1	0	0
01111100	01110000	0	0	0
10001100	11110000	0	1	0
11111100	11110000	0	1	0
00011000	10100001	1	0	0
01000000	10100001	1	0	0

Table 2. Effects of the TM Instruction

As in Test under Mask, the FLAGS control register is the only register affected by this operation. The zero flag (Z) is set if all selected destination bits are 1; it is reset otherwise. The sign flag (S) is set or reset to reflect the result of the AND operation; the overflow flag (V) is always reset. Table 3 illustrates the flag settings which result from the TCM instruction on a variety of source and destination operand combinations. As with the TM instruction, a given TCM instruction will never result in both the Z and S flags being set.

Destination	Source	Flags		
(binary)	(binary)	Z	S	V
10001100	01110000	0	0	0
01111100	01110000	1	0	0
10001100	11110000	0	0	0
11111100	11110000	1	0	0
00011000	10100001	0	1	0
01000000	10100001	0	1	0

Table 3. Effects of the TCM Instruction

Stack Operations

The Z8 stack resides within an area of data memory (internal or external). The current address in the stack is contained in the stack pointer, which decrements as bytes are pushed onto the stack, and increments as bytes are popped from it. The stack pointer occupies two control register bytes (%FE and %FF) in the Z8 register space and may be manipulated like any other register. The stack is useful for subroutine calls, interrupt service routines, and parameter passing and saving. Figure 2 illustrates the downward growth of a stack as bytes are pushed onto it.

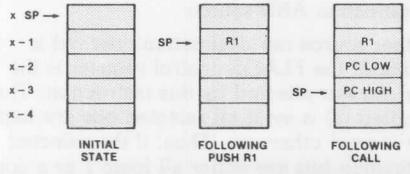


Figure 2. Growth of a Stack

Internal vs. External Stack. The location of the stack in data memory may be selected to be either internal register memory or external data memory. Bit 2 of control register P01M (%F8) controls this selection. Register pair SPH (%FE), SPL (%FF) serves as the stack pointer for an external stack. Register SPL is the stack pointer for an internal stack. In the latter configuration, SPH is available for use as a data register. The following illustrates a code sequence that initializes external stack operations:

```
LD P01M,#%(2)00000000
                !bit 2: select external stack!
LD SPH,#HI STACK
LD SPL,#LO STACK
```

CALL. A subroutine call causes the current Program Counter (the address of the byte following the CALL instruction) to be pushed onto the stack. The Program Counter is loaded with the address specified by the CALL instruction. This address may be a direct address or an indirect register pair reference.

Stack Operations (Continued)

For example,

```
LABEL 1: CALL  %4F98
                !direct addressing: PC is
                 loaded with the hex value
                 4F98;
                 address LABEL 1 + 3 is pushed
                 onto the stack!
LABEL 2: CALL  @RR4
                !indirect addressing: PC is
                 loaded with the contents of
                 working register pair R4, R5;
                 address LABEL 2 + 2 is pushed
                 onto the stack!
LABEL 3: CALL  @%7E
                !indirect addressing: PC is
                 loaded with the contents of
                 register pair %7E, %7F;
                 address LABEL 3 + 2 is pushed
                 onto the stack!
```

RET. The return (RET) instruction causes the top two bytes to be popped from the stack and loaded into the Program Counter. Typically, this is the last instruction of a subroutine and thus restores the PC to the address following the CALL to that subroutine.

Interrupt Machine Cycle. During an interrupt machine cycle, the PC followed by the status flags is pushed onto the stack. (A more detailed discussion of interrupt processing is provided in Section 6.)

IRET. The interrupt return (IRET) instruction causes the top byte to be popped from the stack and loaded into the status flag register, FLAGS (%FC); the next two bytes are then popped and loaded into the Program Counter. In this way, status is restored and program execution continues where it had left off when the interrupt was recognized.

PUSH and POP. The PUSH and POP instructions allow the transfer of bytes between the stack and register memory, thus providing program access to the stack for saving and restoring needed values and passing parameters to subroutines.

Execution of a PUSH instruction causes the stack pointer to be decremented by 1; the operand byte is then loaded into the location pointed to by the decremented stack pointer. Execution of a POP instruction causes the byte addressed by the stack pointer to be loaded into the operand byte; the stack pointer is then incremented by 1. In both cases, the operand byte is designated by either a direct register address or an indirect register reference. For example:

```
PUSH R1     !direct address: push working
             register 1 onto the stack!
POP  5      !direct address: pop the top
             stack byte into register 5!
PUSH @R4    !indirect address: pop the top
             stack byte into the byte
             pointed to by working reg-
             ister 4!
PUSH @17    !indirect address: push onto
             the stack the byte pointed to
             by register 17!
```

Interrupts

The Z8 recognizes six different interrupts from four internal and four external sources, including internal timer/counters, serial I/O, and four Port 3 lines. Interrupts may be individually or globally enabled/disabled via Interrupt Mask Register IMR (%FB) and may be prioritized for simultaneous interrupt resolution via Interrupt Priority Register IPR (%F9). When enabled, interrupt request processing automatically vectors to the designated service routine. When disabled, an interrupt request may be polled to determine when processing is needed.

Interrupt Initialization. Before the Z8 can recognize interrupts following RESET, some initialization tasks must be performed. The initialization routine should configure the Z8 interrupt requests to be enabled/disabled, as required by the target application and assigned a priority (via IPR) for simultaneous enabled-interrupt resolution. An interrupt request is enabled if the corresponding bit in the IMR is set (= 1) and interrupts are globally enabled (bit 7 of IMR = 1). An interrupt request is disabled if the corresponding bit in the IMR is reset (= 0) or interrupts are globally disabled (bit 7 of IMR = 0).

A RESET of the Z8 causes the contents of the Interrupt Request Register IRQ (%FA) to be held to zero until the execution of an EI instruction. Interrupts that occur while the Z8 is in this initial state will not be recognized, since the corresponding IRQ bit cannot be set. The EI instruction is specially decoded by the Z8 to enable the IRQ; simply setting bit 7 of IMR is therefore *not* sufficient to enable interrupt processing following RESET. However, subsequent to this initial EI instruction, interrupts may be globally enabled either by the instruction

```
EI              !enable interrupts!
```

or by a register manipulation instruction such as

```
OR    IMR,#%80
```

To globally disable interrupts, execute the instruction

```
DI              !disable interrupts!
```

This will cause bit 7 of IMR to be reset.

Interrupts *must* be globally disabled prior to any modification of the IMR, IPR or enabled bits of the IRQ (those corresponding to enabled interrupt requests), unless it can be *guaranteed* that an enabled interrupt will not occur during the processing of such instructions. Since interrupts represent the occurrence of events asynchronous to program execution, it is highly unlikely that such a guarantee can be made reliably.

Vectored Interrupt Processing. Enabled interrupt requests are processed in an automatic vectored mode in which the interrupt service routine address is retrieved from within the first 12 bytes of program memory. When an enabled interrupt request is recognized by the Z8, the Program Counter is pushed onto the stack (low order 8 bits first, then high-order 8 bits) followed by the FLAGS register (#%FC). The corresponding interrupt request bit is reset in IRQ, interrupts are globally disabled (bit 7 of IMR is reset), and an indirect jump is taken on the word in location 2x, 2x + 1 (x = interrupt request number, $0 \leq x \leq 5$). For example, if the bytes at addresses %0004 and %0005 contain %05 and %78 respectively, the interrupt machine cycle for IRQ2 will cause program execution to continue at address %0578.

When interrupts are sampled, more than one interrupt may be pending. The Interrupt Priority Register (IPR) controls the selection of the pending interrupt with highest priority. While this interrupt is being serviced, a higher-priority interrupt may occur. Such interrupts may be allowed service within the current interrupt service routine (nested) or may be held until the current service routine is complete (non-nested).

To allow nested interrupt processing, interrupts must be selectively enabled upon entry to an interrupt service routine. Typically, only higher-priority interrupts would be allowed to nest within the current interrupt service. To do this, an interrupt routine must "know" which interrupts have a higher priority than the current interrupt request. Selection of such nesting priorities is usually a reflection of the priorities established in the Interrupt Priority Register (IPR). Given this data, the first instructions executed in the service routine should be to save the current Interrupt Mask Register, mask off all interrupts of lower and equal priority, and globally enable interrupts

(EI). For example, assume that service of interrupt requests 4 and 5 are nested within the service of interrupt request 3. The following illustrates the code required to enable IRQ4 and IRQ5:

```
CONSTANT
        INT_MASK_3          :=      %(2) 00110000
GLOBAL
IRQ3_service        PROCEDURE       ENTRY
!service routine for IRQ3!
        PUSH IMR                    !save Interrupt Mask Register!
                !interrupts were globally disabled during the interrupt
                 machine cycle - no DI is needed prior to modification of IMR!
        AND  IMR,#INT_MASK_3        !disable all but IRQ4 & 5!
        EI
        !...!       !service interrupt!
                    !interrupts are globally enabled now — must disable them prior to
                     modification of IMR!
        DI
        POP  IMR                    !restore entry IMR!
        IRET
END IRQ3_service
```

Note that IRQ4 and IRQ5 are enabled by the above sequence only if their respective IMR bits = 1 on entry to IRQ3_service.

The service routine for an interrupt whose processing is to be completed without interruption should not allow interrupts to be nested within it. Therefore, it need not modify the IMR, since interrupts are disabled automatically during the interrupt machine cycle.

The service routine for an enabled interrupt is typically concluded with an IRET instruction, which restores the FLAGS register and Program Counter from the top of the stack and globally enables interrupts. To return from an interrupt service routine without re-enabling interrupts, the following code sequence could be used:

```
POP  FLAGS
        !FLAGS ← @SP!
RET     !PC ← @SP!
```

This accomplishes all the functions of IRET, except that IMR is not affected.

Polled Interrupt Processing Disabled interrupt requests may be processed in a polled mode, in which the corresponding bits of the Interrupt Request Register (IRQ) are examined by the software. When an interrupt request bit is found to be a logic 1, the interrupt should be processed by the appropriate service routine. During such processing, the interrupt request bit in the IRQ must be cleared by the software in order for subsequent interrupts on that line to be distinguished from the current one. If more than one interrupt request is to be processed in a polled mode, polling should occur in the order of established priorities. For example, assume that IRQ0, IRQ1, and IRQ4 are to be polled and that established priorities are, from high to low, IRQ4, IRQ0, IRQ1. An instruction sequence like the following should be used to poll and service the interrupts:

Interrupts (Continued)

```
!...!
 !poll interrupt inputs here!
            TCM     IRQ, #%(2)00010000      !IRQ4 need service?!
            JR      NZ, TEST0               !no!
            CALL    IRQ4_service            !yes!
TEST0:      TCM     IRQ, #%(2)00000001      !IRQ0 need service?!
            JR      NZ, TEST1               !no!
            CALL    IRQ0_service            !yes!
TEST1:      TCM     IRQ, #%(2)00000010      !IRQ1 need service?!
            JR      NZ, DONE                !no!
            CALL    IRQ1_service            !yes!
DONE:       !...!

IRQ4_service        PROCEDURE       ENTRY
            !...!
            AND     IRQ, #%(2)11101111      !clear IRQ4!
            !...!
            RET
END IRQ4_service

IRQ0_service        PROCEDURE       ENTRY
            !...!
            AND     IRQ, #%(2)11111110      !clear IRQ0!
            !...!
            RET
END IRQ0_service

IRQ1_service        PROCEDURE       ENTRY
            !...!
            AND     IRQ, #%(2)11111101      !clear IRQ1!
            !...!
            RET
END IRQ1_service
!...!
```

Timer/Counter Functions

The Z8 provides two 8-bit timer/counters, T_0 and T_1, which are adaptable to a variety of application needs and thus allow the software (and external hardware) to be relieved of the bulk of such tasks. Included in the set of such uses are:

- Interval delay timer
- Maintenance of a time-of-day clock
- Watch-dog timer
- External event counting
- Variable pulse train output
- Duration measurement of external event
- Automatic delay following external event detection

Each timer/counter is driven by its own 6-bit prescaler, which is in turn driven by the internal Z8 clock divided by four. For T_1, the internal clock may be gated or triggered by an external event or may be replaced by an external clock input. Each timer/counter may operate in either single-pass or continuous mode where, at end-of-count, either counting stops or the counter reloads and continues counting. The counter and prescaler registers may be altered individually while the timer/counter is running; the software controls whether the new values are loaded immediately or when end-of-count (EOC) is reached.

Although the timer/counter prescaler registers (PRE0 and PRE1) are write-only, there is a technique by which the timer/

Timer/Counter Functions (Continued)

counters may simulate a readable prescaler. This capability is a requirement for high resolution measurement of an event's duration. The basic approach requires that one timer/counter be initialized with the desired counter and prescaler values. The second timer/counter is initialized with a counter equal to the prescaler of the first timer/counter and a prescaler of 1. The second timer/counter must be programmed for continuous mode. With both timer/counters driven by the internal clock and started and stopped simultaneously, they will run synchronous to one another; thus, the value read from the second counter will always be equivalent to the prescaler of the first.

Time/Count Interval Calculation To determine the time interval (i) until EOC, the equation

$$i = t \times p \times v$$

characterizes the relation between the prescaler (p), counter (v), and clock input period (t); t is given by

$$1/(XTAL/8)$$

where XTAL is the Z8 input clock frequency; p is in the range 1 – 64; v is in the range 1 – 256. When programming the prescaler and counter registers, the maximum load value is truncated to six and eight bits, respectively, and is therefore programmed as zero. For an input clock frequency of 8 MHz, the prescaler and counter register values may be programmed to time an interval in the range

$$1\ \mu s \times 1 \times 1 \leq i \leq 1\ \mu s \times 64 \times 256$$

$$1\ \mu s \leq i \leq 16.384\ ms$$

To determine the count (c) until EOC for T_1 with external clock input, the equation

$$c = p \times v$$

characterizes the relation between the T_1 prescaler (p) and the T_1 counter (v). The divide-by-8 on the input frequency is bypassed in this mode. The count range is

$$1 \times 1 \leq c \leq 64 \times 256$$

$$1 \leq c \leq 16{,}384$$

T_{OUT} Modes. Port 3, bit 6 ($P3_6$) may be configured as an output (T_{OUT}) which is dynamically controlled by one of the following:

- T_0
- T_1
- Internal clock

When driven by T_0 or T_1, T_{OUT} is reset to a logic 1 when the corresonding load bit is set in timer control register TMR (%F1) and toggles on EOC from the corresponding counter. When T_{OUT} is driven by the internal clock, that clock is directly output on $P3_6$.

While programmed as T_{OUT}, $P3_6$ is disabled from being modified by a write to port register %03; however, its current output may be examined by the Z8 software by a read to port register %03.

T_{IN} Modes. Port 3, bit 1 ($P3_1$) may be configured as an input (T_{IN}) which is used in conjunction with T_1 in one of four modes:

- External clock input
- Gate input for internal clock
- Nonretriggerrable input for internal clock
- Retriggerable input for internal clock

For the latter two modes, it should be noted that the existence of a synchronizing circuit within the Z8 causes a delay of two to three internal clock periods following an external trigger before clocking of the counter actually begins.

Each High-to-Low transition on T_{IN} will generate interrupt request IRQ2, regardless of the selected T_{IN} mode or the enabled/disabled state of T_1. IRQ2 must therefore be masked or enabled according to the needs of the application.

The "external clock input" T_{IN} mode supports the counting of external events, where an event is seen as a High-to-Low transition on T_{IN}. Interrupt request IRQ5 is generated on the n^{th} occurrence (single-pass mode) or on every n^{th} occurrence (continuous mode) of that event.

Timer/Counter Functions (Continued)

The "gate input for internal clock" T_{IN} mode provides for duration measurement of an external event. In this mode, the T_1 prescaler is driven by the Z8 internal clock, gated by a High level on T_{IN}. In other words, T_1 will count while T_{IN} is High and stop counting while T_{IN} is Low. Interrupt request IRQ2 is generated on the High-to-Low transition on T_{IN}. Interrupt request IRQ5 is generated on T_1 EOC. This mode may be used when the width of a High-going pulse needs to be measured. In this mode, IRQ2 is typically the interrupt request of most importance, since it signals the end of the pulse being measured. If IRQ5 is generated prior to IRQ2 in this mode, the pulse width on T_{IN} is too large for T_1 to measure in a single pass.

The "nonretriggerable input" T_{IN} mode provides for automatic delay timing following an external event. In this mode, T_1 is loaded and clocked by the Z8 internal clock following the first High-to-Low transition on T_{IN} after T_1 is enabled. T_{IN} transitions that occur after this point do not affect T_1. In single-pass mode, the enable bit is reset on EOC; further T_{IN} transitions will not cause T_1 to load and begin counting until the software sets the enable bit again. In continuous mode, EOC does not modify the enable bit, but the counter is reloaded and counting continues immediately; IRQ5 is generated every EOC until software resets the enable bit. This T_{IN} mode may be used, for example, to time the line feed delay following end of line detection on a printer or to delay data sampling for some length of time following a sample strobe.

The "retriggerable input" T_{IN} mode will load and clock T_1 with the Z8 internal clock on every occurrence of a High-to-Low transition on T_{IN}. T_1 will time-out and generate interrupt request IRQ5 when the programmed time interval (determined by T_1 prescaler and load register values) has elapsed since the last High-to-Low transition on T_{IN}. In single-pass mode, the enable bit is reset on EOC; further T_{IN} transitions will not cause T_1 to load and begin counting until the software sets the enable bit again. In continuous mode, EOC does not modify the enable bit, but the counter is reloaded and counting continues immediately; IRQ5 is generated at every EOC until the software resets the enable bit. This T_{IN} mode may provide such functions as watch-dog timer (e.g., interrupt if conveyor belt stopped or clock pulse missed), or keyboard time-out (e.g., interrupt if no input in x ms).

Examples. Several possible uses of the timer/counters are given in the following four examples.

Time of Day Clock. The following module illustrates the use of T_1 for maintenance of a time of day clock, which is kept in binary format in terms of hours, minutes, seconds, and hundredths of a second. It is desired that the clock be updated once every hundredth of a second; therefore, T_1 is programmed in continuous mode to interrupt 100 times a second. Although T_1 is used for this example, T_0 is equally suited for the task.

The procedure for initializing the timer (TOD_INIT), the interrupt service routine (TOD) which updates the clock, and the interrupt vector for T_1 end-of-count (IRQ_5) are illustrated below. XTAL = 7.3728 MHz is assumed.

```
Z8ASM     2.0
LOC    OBJ CODE   STMT SOURCE STATEMENT

                     1 TIMER1  MODULE
                     2 CONSTANT
                     3  HOUR   :=      R12
                     4  MINUTE :=      R13
                     5  SECOND :=      R14
                     6  HUND   :=      R15
                     7         $SECTION PROGRAM
                     8 GLOBAL
                     9 !IRQ5 interrupt vector!
                    10         $ABS    10
P 0000 000F'         11 IRQ_5   ARRAY   [1 WORD]  :=    [TOD]
                    12
                    13         $REL
P 000C              14 TOD_INIT        PROCEDURE
                    15 ENTRY
```

Timer/Counter Functions (Continued)

```
P 0000 E6  F3  93     16          LD      PRE1,#%(2)10010011
                      17                  !bit 2-7: prescaler = 36,
                      18                   bit 1: internal clock;
                      19                   bit 0: continuous mode!
P 0003 E6  F2  00     20          LD      T1,#0       !(256) time-out =
                      21                   1/100 second!
P 0006 46  F1  0C     22          OR      TMR,#%0C    !load, enable T1!
P 0009 8F             23          DI
P 000A 46  FB  20     24          OR      IMR,#%20    !enable T1 interrupt!
P 000D 9F             25          EI
P 000E AF             26          RET
P 000F                27 END      TOD_INIT
                      28
P 000F                29 TOD      PROCEDURE
                      30 ENTRY
P 000F 70  FD         31          PUSH    RP
                      32 !Working register file %10 to %1F contains
                      33  the time of day clock!
P 0011 31  10         34          SRP     #%10
P 0013 FE             35          INC     HUND                !1 more .01 sec!
P 0014 A6  EF  64     36          CP      HUND,#100           !full second yet?!
P 0017 EB  13         37          JR      NE,TOD_EXIT         !jump if no!
P 0019 B0  EF         38          CLR     HUND
P 001B EE             39          INC     SECOND              !1 more second!
P 001C A6  EE  3C     40          CP      SECOND,#60          !full minute yet?!
P 001F EB  0B         41          JR      NE,TOD_EXIT         !jump if no!
P 0021 B0  EE         42          CLR     SECOND
P 0023 DE             43          INC     MINUTE              !1 more minute!
P 0024 A6  ED  3C     44          CP      MINUTE,#60          !full hour yet?!
P 0027 EB  03         45          JR      NE,TOD_EXIT         !jump if no!
P 0029 B0  ED         46          CLR     MINUTE
P 002B CE             47          INC     HOUR
                      48 TOD_EXIT:
P 002C 50  FD         49          POP     RP                  !restore entry RP!
P 002E BF             50          IRET
P 002F                51 END      TOD
                      52 END      TIMER1

   0 ERRORS
ASSEMBLY COMPLETE
```

TOD_INIT:
7 instructions
15 bytes
16 µs

TOD:
17 instruction
32 bytes
19.5 µs (average) including interrupt response time

Variable Frequency, Variable Pulse Width Output. The following module illustrates one possible use of T_{OUT}. Assume it is necessary to generate a pulse train with a 10% duty cycle, where the output is repetitively high for 1.6 ms and then low for 14.4 ms. To do this, T_{OUT} is controlled by end-of-count from T_1, although T_0 could alternately be chosen. This example makes use of the Z8 feature that allows a timer's counter register to be modified without disturbing the count in progress. In continuous mode, the new value is loaded when T_1 reaches EOC. T_1 is first loaded and enabled with values to generate the short interval. The counter register is then immediately modified with the value to generate the long interval; this value is loaded into the counter automatically on T_1 EOC. The prescaler selected value must be the same for both long and short intervals. Note that the initial loading of the T_1 counter register is followed by setting the T_1 load bit of timer control register TMR (%F1); this action causes T_{OUT} to be reset to a logic 1 output. Each subsequent modification of the T_1 counter register does not affect the current T_{OUT} level, since the T_1 load bit is NOT altered by the software. The new value is loaded on EOC, and T_{OUT} will toggle at that time. The T_1 interrupt service routine should simply modify the

Timer/Counter Functions (Continued)

T_1 counter register with the new value, alternating between the long and short interval values.

In the example which follows, bit 0 of register %04 is used as a software flag to indicate which value was loaded last. This module illustrates the procedure for T_1/T_{OUT} initialization (PULSE_INIT), the T_1 interrupt service routine (PULSE), and the interrupt vector for T_1 EOC (IRQ_5). XTAL = 8 MHz is assumed.

```
Z8ASM     2.0
LOC     OBJ CODE    STMT SOURCE STATEMENT

                        1 TIMER2  MODULE
                        2         $SECTION PROGRAM
                        3 GLOBAL
                        4 !IRQ5 interrupt vector!
                        5         $ABS    10
P 0000 0017'            6 IRQ_5   ARRAY   [1 WORD]  :=    [PULSE]
                        7
                        8         $REL
P 000C                  9 PULSE_INIT      PROCEDURE
                       10 ENTRY
P 0000 E6  F3  03      11         LD      PRE1,#%(2)00000011
                       12                         !bit 2-7: prescaler = 64;
                       13                          bit 1: internal clock;
                       14                          bit 0: continuous mode!
P 0003 E6  F7  00      15         LD      P3M,#00   !bit 5: let P36 be Tout!
P 0006 E6  F2  19      16         LD      T1,#25           !for short interval!
P 0009 8F              17         DI
P 000A 46  FB  20      18         OR      IMR,#%(2)00100000  !enable T1 interrupt!
P 000D E6  F1  8C      19         LD      TMR,#%(2)10001100
                       20                         !bit 6-7: Tout controlled
                       21                                    by T1;
                       22                          bit 3: enable T1;
                       23                          bit 2: load T1 !
                       24 !Set long interval counter, to be loaded on T1 EOC!
P 0010 E6  F2  E1      25         LD      T1,#225
                       26 !Clear alternating flag for PULSE!
P 0013 B0  04          27         CLR     %04              != 0 : 25 next;
                       28                                   = 1 : 225 next !
P 0015 9F              29         EI
P 0016 AF              30         RET
P 0017                 31 END     PULSE_INIT
                       32
                       33
P 0017                 34 PULSE   PROCEDURE
                       35 ENTRY
P 0017 E6  F2  E1      36         LD      T1,#225          !new load value!
P 001A B6  04  01      37         XOR     %04,#1           !which value next?!
P 001D 6B  03          38         JR      Z,PULSE_EXIT     !should be 225!
P 001F E6  F2  19      39         LD      T1,#25           !should be 25!
                       40 PULSE_EXIT:
P 0022 BF              41         IRET
P 0023                 42 END     PULSE
                       43 END     TIMER2

   0 ERRORS
ASSEMBLY COMPLETE
```

PULSE_INIT:
10 instructions
23 bytes
23 µs

PULSE:
5 instructions
12 bytes
25 µs (average) including interrupt response time

Timer/Counter Functions (Continued)

Cascaded Timer/Counters. For some applications it may be necessary to measure a greater time interval than a single timer/counter can measure (16.384 ms). In this case, T_{IN} and T_{OUT} may be used to cascade T_0 and T_1 to function as a single unit. T_{OUT}, programmed to toggle on T_0 end-of-count, should be wired back to T_{IN}, which is selected as the external clock input for T_1. With T_0 programmed for continuous mode, T_{OUT} (and therefore T_{IN}) goes through a High-to-Low transition (causing T_1 to count) on every other T_0 EOC. Interrupt request IRQ5 is generated when the programmed time interval has elapsed. Interrupt requests IRQ2 (generated on every T_{IN} High-to-Low transition) and IRQ4 (generated on T_0 EOC) are of no importance in this application and are therefore disabled.

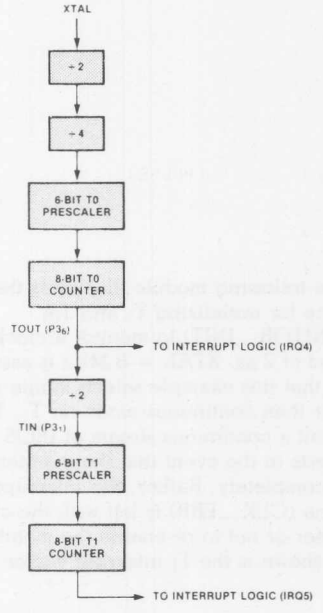


Figure 3. Cascaded Timer/Counters

To determine the time interval (i) until EOC, the equation

$$i = t \times p0 \times v0 \times (2 \times p1 \times v1 - 1)$$

characterizes the relation between the T_0 prescaler (p0) and counter (v0), the T_1 prescaler (p1) and counter (v1), and the clock input period (t); t is defined in Section 7.1. Assuming XTAL = 8 MHz, the measurable time interval range is

$$1\ \mu s \times 1 \times 1 \times (2 \times 1 - 1) \leq i \leq 1\ \mu s \times 64 \times 256 \times (2 \times 64 \times 256 - 1)$$

$$1\ \mu s \leq i \leq 536.854528\ s$$

Figure 3 illustrates the interconnection between T_0 and T_1. The following module illustrates the procedure required to initialize the timers for a 1.998 second delay interval:

```
Z8ASM      2.0
LOC     OBJ CODE    STMT SOURCE STATEMENT

                      1 TIMER3  MODULE
                      2 GLOBAL
P 0000                3 TIMER_16          PROCEDURE
                      4 ENTRY
P 0000 E6  F3  28     5          LD       PRE1,#%(2)00101000
                      6                           !bit 2-7: prescaler = 10;
                      7                            bit 1: external clock;
                      8                            bit 0: single-pass mode!
P 0003 E6  F7  00     9          LD       P3M,#00    !bit 5: let P36 be Tout!
P 0006 E6  F2  64    10          LD       T1,#100          !T1 counter register!
P 0009 E6  F5  29    11          LD       PRE0,#%(2)00101001
                     12                           !bit 2-7: prescaler = 10;
                     13                            bit 0: continuous mode!
P 000C E6  F4  64    14          LD       T0,#100          !T0 counter register!
P 000F 8F            15          DI
P 0010 56  FB  2B    16          AND      IMR,#%(2)00101011  !disable IRQ2 (Tin);
                     17                                        and IRQ4 (T0) !
P 0013 46  FB  20    18          OR       IMR,#%(2)00100000  !enable IRQ5 (T1)!
P 0016 9F            19          EI
P 0017 E6  F1  4F    20          LD       TMR,#%(2)01001111
```

Timer/Counter Functions (Continued)

```
                        21                          !bit 6-7: Tout controlled
                        22                                    by T0;
                        23                           bit 4-5: Tin mode is ext.
                        24                                    clock input;
                        25                           bit 3: enable T1;
                        26                           bit 2: load T1;
                        27                           bit 1: enable T0;
                        28                           bit 0: load T0 !
P 001A AF               29          RET
P 001B                  30 END      TIMER_16
                        31 END      TIMER3

   0 ERRORS
ASSEMBLY COMPLETE

 11 instructions
 27 bytes
 26.5 µs
```

Clock Monitor. T_1 and T_{IN} may be used to monitor a clock line (in a diskette drive, for example) and generate an interrupt request when a clock pulse is missed. To accomplish this, the clock line to be monitored is wired to $P3_1$ (T_{IN}). T_{IN} should be programmed as a retriggerable input to T_1, such that each falling edge on T_{IN} will cause T_1 to reload and continue counting. If T_1 is programmed to time-out after an interval of one-and-a-half times the clock period being monitored, T_1 will time-out and generate interrupt request IRQ5 only if a clock pulse is missed.

The following module illustrates the procedure for initializing T_1 and T_{IN} (MONITOR_INIT) to monitor a clock with a period of 2 µs. XTAL = 8 MHz is assumed. Note that this example selects single-pass rather than continuous mode for T_1. This is to prevent a continuous stream of IRQ5 interrupt requests in the event that the monitored clock fails completely. Rather, the interrupt service routine (CLK_ERR) is left with the choice of whether or not to re-enable the monitoring. Also shown is the T_1 interrupt vector (IRQ_5).

```
Z8ASM     2.0
LOC    OBJ CODE   STMT SOURCE STATEMENT

                    1 TIMER4  MODULE
                    2         $SECTION PROGRAM
                    3 GLOBAL
                    4 !IRQ5 interrupt vector!
                    5         $ABS    10
P 0000 0015'        6 IRQ_5   ARRAY   [1 WORD]  :=    [CLK_ERR]
                    7
                    8         $REL
P 000C              9 MONITOR_INIT    PROCEDURE
                   10 ENTRY
P 0000 E6  F3  04  11         LD      PRE1,#%(2)00000100
                   12                                !bit 2-7: prescaler = 1;
                   13                                 bit 1: external clock;
                   14                                 bit 0: single-pass mode!
P 0003 E6  F7  00  15         LD      P3M,#00    !bit 5: let P36 be Tout!
P 0006 E6  F2  03  16         LD      T1,#3            !T1 load register,
                   17                                  = 1.5 * 2 usec   !
P 0009 8F          18         DI
P 000A 56  FB  3B  19         AND     IMR,#%(2)00111011  !disable IRQ2 (Tin)!
P 000D 46  FB  20  20         OR      IMR,#%(2)00100000  !enable IRQ5 (T1)!
P 0010 9F          21         EI
                   22
```

Timer/Counter Functions (Continued)

```
P 0011 E6  F1  38     23          LD      TMR,#%(2)00111000
                      24                              !bit 4-5: Tin mode is
                      25                                        retrig. input;
                      26                               bit 3: enable T1 !
P 0014 AF             27          RET
P 0015                28 END      MONITOR_INIT
                      29
                      30
P 0015                31 CLK_ERR  PROCEDURE
                      32 ENTRY
                      33          !...!              !handle the missed clock!
                      34
                      35 !if clock monitoring should continue...!
P 0015 46  F1  08     36          OR      TMR,#%(2)00001000
                      37                              !bit 3: enable T1 !
P 0018 BF             38          IRET
P 0019                39 END      CLK_ERR
                      40 END      TIMER4

    0 ERRORS
ASSEMBLY COMPLETE
```

MONITOR_INIT:
9 instructions
21 bytes
21.5 µs

CLK_ERR:
2 + instructions
4 + bytes
18.5 + µs including interrupt response time

I/O Functions

The Z8 provides 32 I/O lines mapped into registers 0-3 of the internal register file. Each nibble of port 0 is individually programmable as input, output, or address/data lines (A_{15}-A_{12}, A_{11}-A_8). Port 1 is programmable as a single entity to provide input, output, or address/data lines (AD_7-AD_0). The operating modes for the bits of Ports 0 and 1 are selected by control register P01M (%F8). Selection of I/O lines as address/data lines supports access to external program and data memory; this is discussed in Section 3. Each bit of Port 2 is individually programmable as an input or an output bit. Port 2 bits programmed as outputs may also be programmed (via bit 0 of P3M) to all have active pull-ups or all be open-drain (active pull-ups inhibited). In Port 3, four bits ($P3_0$-$P3_3$) are fixed as inputs, and four bits ($P3_4$-$P3_7$) are fixed as outputs, but their functions are programmable. Special functions provided by Port 3 bits are listed in Table 4. Use of the Data Memory select output is discussed in Section 3; uses of T_{IN} and T_{OUT} are discussed in Section 7.

Function	Bit	Signal
Handshake	$P3_1$	$\overline{DAV2}$/RDY2
	$P3_2$	$\overline{DAV0}$/RDY0
	$P3_3$	$\overline{DAV1}$/RDY1
	$P3_4$	RDY1/$\overline{DAV1}$
	$P3_5$	RDY0/$\overline{DAV0}$
	$P3_6$	RDY2/$\overline{DAV2}$
Interrupt Request	$P3_0$	IRQ3
	$P3_1$	IRQ2
	$P3_2$	IRQ0
	$P3_3$	IRQ1
Counter/Timer	$P3_1$	T_{IN}
	$P3_6$	T_{OUT}
Data Memory Select	$P3_4$	$\overline{DM}$
Status Out		
Serial I/O	$P3_0$	Serial In
	$P3_7$	Serial Out

Table 4. Port 3 Special Functions

Asynchronous Receiver/Transmitter Operation. Full-duplex, serial asynchronous receiver/transmitter operation is provided by the Z8 via $P3_7$ (output) and $P3_0$ (input) in conjunction with control register SIO (%F0), which is actually two registers: receiver buffer and transmitter buffer. Counter/Timer T_0 provides the clock for control of the bit rate.

The Z8 always receives and transmits eight bits between start and stop bits. However, if parity is enabled, the eighth bit (D_7) is replaced by the odd-parity bit when transmitted and a parity-error flag (= 1 if error)

I/O Functions (Continued)

Character Loaded Into SIO	Transmitted To Serial Line	Received From Serial Line	Character Transferred To SIO	Note*
11000011	01000011	01000011	01000011	no error
11000011	01000011	01000111	11000111	error
01111000	11111000	11111000	01111000	no error
01111000	11111000	01111000	11111000	error

Table 5. Serial I/O With Odd Parity

* Left-most bit is D7

when received. Table 5 illustrates the state of the parity bit/parity error flag during serial I/O with parity enabled.

Although the Z8 directly supports either odd parity or no parity for serial I/O operation, even parity may also be provided with additional software support. To receive and transmit with even parity, the Z8 should be configured for serial I/O with odd parity disabled. The Z8 software must calculate parity and modify the eighth bit prior to the load of a character into SIO and then modify a parity error flag following the load of a character from SIO. All other processing required for serial I/O (e.g. buffer management, error handling, etc.) is the same as that for odd parity operations.

To configure the Z8 for Serial I/O, it is necessary to:

- Enable $P3_0$ and $P3_7$ for serial I/O and select parity,
- Set up T_0 for the desired bit rate,
- Configure IRQ3 and IRQ4 for polled or automatic interrupt mode,
- Load and enable T_0.

To enable $P3_0$ and $P3_7$ for serial I/O, bit 6 of P3M (R247) is set. To enable odd parity, bit 7 of P3M is set; to disable it, the bit is reset. For example, the instruction

```
LD      P3M,#%40
```

will enable serial I/O, but disable parity. The instruction

```
LD      P3M,#%C0
```

will enable serial I/O, and enable odd parity.

In the following discussions, bit rate refers to all transmitted bits, including start, stop, and parity (if enabled). The serial bit rate is given by the equation:

$$\text{bit rate} = \frac{\text{input clock frequency}}{(2 \times 4 \times T_0 \text{ prescaler} \times T_0 \text{ counter} \times 16)}$$

The final divide-by-16 is incurred for serial communications, since in this mode T_0 runs at 16 times the bit rate in order to synchronize the data stream. To configure the Z8 for a specific bit rate, appropriate values must first be selected for T_0 prescaler and T_0 counter by the above equation; these values are then programmed into registers T_0 (%F4) and PRE0 (%F5) respectively. Note that PRE0 also controls the continuous vs. single-pass mode for T_0; continuous mode should be selected for serial I/O. For example, given an input clock frequency of 7.3728 MHz and a selected bit rate of 9600 bits per second, the equation is satisfied by T_0 counter = 2 and prescaler = 3. The following code sequence will configure the T_0 counter and T_0 prescaler registers:

```
LD   T0,#2     !T0 counter = 2!
LD   PRE0,#%(2)00001101
                !bit 2-7: prescaler = 3; bit 0:
                 continuous mode!
```

Interrupt request 3 (IRQ3) is generated whenever a character is transferred into the receive buffer; interrupt request 4 (IRQ4) is generated whenever a character is transferred out of the transmit buffer. Before accepting such interrupt requests, the Interrupt Mask, Request, and Priority Registers (IMR, IRQ, and IPR) must be programmed to configure the mode of interrupt response. The section on Interrupt Processing provides a discussion of interrupt configurations.

To load and enable T_0, set bits 0 and 1 of the timer mode register (TMR) via an instruction such as

```
OR   TMR,#%03
```

This will cause the T_0 prescaler and counter registers (PRE0 and T_0) to be transferred to the T_0 prescaler and counter. In addition, T_0 is enabled to count, and serial I/O operations will commence.

I/O Functions (Continued)

Characters to be output to the serial line should be written to serial I/O register SIO (%F0). IRQ4 will be generated when all bits have been transferred out.

Characters input from the serial line may be read from SIO. IRQ3 will be generated when a full character has been transferred into SIO.

The following module illustrates the receipt of a character and its immediate echo back to the serial line. It is assumed that the Z8 has been configured for serial I/O as described above, with IRQ3 (receive) enabled to interrupt, and IRQ4 (transmit) configured to be polled. The received character is stored in a circular buffer in register memory from address %42 to %5F. Register %41 contains the address of the next available buffer position and should have been initialized by some earlier routine to #%42.

```
Z8ASM      2.0
LOC     OBJ CODE    STMT SOURCE STATEMENT

                      1 SERIAL_IO       MODULE
                      2 CONSTANT
                      3  next_addr      :=      %41
                      4  start          :=      %42
                      5  length         :=      %1E
                      6 $SECTION PROGRAM
                      7 GLOBAL
                      8 !IRQ3 vector!
                      9         $ABS    6
P 0006 0000'          10 IRQ_3    ARRAY [1 WORD] :=  [GET_CHARACTER]
                     11
                     12         $REL    0
P 0000                13 GET_CHARACTER    PROCEDURE       ENTRY
                     14
                     15 !Serial I/O receive interrupt service!
                     16 !Echo received character and wait for
                     17  echo completion!
P 0000 E4  F0  F0    18         ld      SIO,SIO          !echo!
                     19
                     20 !save it in circular buffer!
P 0003 F5  F0  41    21         ld      @next_addr,SIO   !save in buffer!
P 0006 20  41        22         inc     next_addr        !point to next position!
P 0008 A6  41  60    23         cp      next_addr,#start+length
                     24                                  !wrap-around yet?!
P 000B EB  03        25         jr      ne,echo_wait     !no.!
P 000D E6  41  42    26         ld      next_addr,#start !yes. point to start!
                     27 !now, wait for echo complete!
                     28 echo_wait:
P 0010 66  FA  10    29         tcm     IRQ,#%10         !transmitted yet?!
P 0013 EB  FB        30         jr      nz,echo_wait     !not yet!
                     31
P 0015 56  FA  EF    32         and     IRQ,#%EF         !clear IRQ4!
P 0018 BF            33         IRET                     !return from interrupt!
P 0019               34 END      GET_CHARACTER
                     35 END      SERIAL_IO

   0 ERRORS
ASSEMBLY COMPLETE
```

10 instructions
25 bytes
35.5 µs + 5.5 µs for each additional pass through the echo_wait loop, including interrupt response time

I/O Functions (Continued)

Automatic Bit Rate Detection. In a typical system, where serial communication is required (e.g. system with a terminal), the desired bit rate is either user-selectable via a switch bank or nonvariable and "hard-coded" in the software. As an alternate method of bit-rate detection, it is possible to automatically determine the bit rate of serial data received by measuring the length of a start bit. The advantage of this method is that it places no requirements on the hardware design for this function and provides a convenient (automatic) operator interface.

In the technique described here, the serial channel of the Z8 is initialized to expect a bit rate of 19,200 bits per second. The number of bits (n) received through Port pin P30 for each bit transmitted is expressed by

$$n = 19{,}200/b$$

where b = transmission bit rate. For example, if the transmission bit rate were 1200 bits per second, each incoming bit would appear to the receiving serial line as 19,200/1200 or 16 bits.

The following example is capable of distinguishing between the bit rates shown in Table 6 and assumes an input clock frequency of 7.3728 MHz, a T_0 prescaler of 3, and serial I/O enabled with parity disabled. This example requires that a character with its low order bit = 1 (such as a carriage return) be sent to the serial channel. The start bit of this character can be measured by counting the number of zero bits collected before the low order 1 bit. The number of zero bits actually collected into data bits by the serial channel is less than n (as given in the above equation), due to the detection of start and stop bits. Figure 4 illustrates the collection (at 19,200

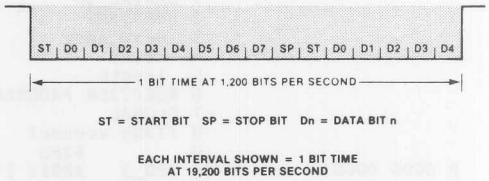


Figure 4. Collection of a Start Bit Transmitted at 1,200 BPS and Received at 19,200 BPS

Bit Rate	Number of Bits Received Per Bit Transmitted	Number of 0 Bits Collected as Data Bits		T_0 Counter	
		dec	binary	dec	binary
19200	1	0	00000000	1	00000001
9600	2	1	00000001	2	00000010
4800	4	3	00000011	4	00000100
2400	8	7	00000111	8	00001000
1200	16	13	00001101	16	00010000
600	32	25	00011001	32	00100000
300	64	49	00110001	64	01000000
150	128	97	01100001	128	10000000

Table 6. Inputs to the Automatic Bit Rate Detection Algorithm

bits per second) of a zero bit transmitted to the Z8 at 1,200 bits per second. Notice that only 13 of the 16 zero bits received are collected as data bits.

Once the number of zero bits in the start bit has been collected and counted, it remains to translate this count into the appropriate T_0 counter value and program that value into T_0 (%F4). The patterns shown in the two binary columns of Table 6 are utilized in the algorithm for this translation.

As a final step, if incoming data is to commence immediately, it is advisable to wait until the remainder of the current "elongated" character has been received, thus "flushing" the serial line. This can be accomplished either via a software loop, or by programming T_1 to generate an interrupt request after the appropriate amount of time has elapsed. Since a character is composed of eight bits plus a minimum of one stop bit following the start bit, the length of time to delay may be expressed as

$$(9 \times n)/b$$

where n and b are as defined above. The following module illustrates a sample program for automatic bit rate detection.

I/O Functions (Continued)

```
Z8ASM      2.0
LOC    OBJ CODE   STMT SOURCE STATEMENT

                    1 bit_rate        MODULE
                    2 EXTERNAL
                    3 DELAY   PROCEDURE
                    4 GLOBAL
P 0000              5 main    PROCEDURE
                    6         ENTRY
P 0000 8F           7         di                  !disable interrupts!
P 0001 56  FB  77   8         and     IMR,#%77    !IRQ3 polled mode!
P 0004 56  FA  F7   9         and     IRQ,#%F7    !clear IRQ3!
P 0007 E6  F7  40  10         ld      P3M,#%40    !enable serial I/O!
P 000A E6  F4  01  11         ld      T0,#1
P 000D E6  F5  0D  12         ld      PRE0,#(3 SHL 2)+1  !bit rate = 19,200;
                   13                                    continuous count mode!
P 0010 B0  E0      14         clr     R0          !init. zero byte counter!
P 0012 E6  F1  03  15         ld      TMR,#3      !load and enable T0!
                   16
                   17 !collect input bytes by counting the number of null
                   18  characters received.  Stop when non-zero byte received!
                   19 collect:
P 0015 76  FA  08  20         TM      IRQ,#%08    !character received?!
P 0018 6B  FB      21         jr      z,collect   !not yet!
P 001A 18  F0      22         ld      R1,SIO      !get the character!
P 001C 56  FA  F7  23         and     IRQ,#%F7    !clear interrupt request!
P 001F 1E          24         inc     R1          !compare to 0 ...!
P 0020 1A  05      25         djnz    R1,bitloop !...(in 3 bytes of code)!
P 0022 06  E0  08  26         add     R0,#8       !update count of 0 bits!
P 0025 8B  EE      27         jr      collect
                   28 bitloop:                    !add in zero bits from low
                   29                              end of 1st non-zero byte!
P 0027 E0  E1      30         RR      R1
P 0029 7B  03      31         jr      c,count_done
P 002B 0E          32         inc     R0
P 002C 8B  F9      33         jr      bitloop
                   34
                   35 !R0 has number of zero bits collected!
                   36 !translate R0 to the appropriate T0 counter value!
                   37 count_done:                 !R0 has count of zero bits!
P 002E 1C  07      38         ld      R1,#7
P 0030 2C  80      39         ld      R2,#%80     !R2 will have T0 counter value!
P 0032 90  E0      40         RL      R0
                   41
P 0034 90  E0      42 loop:   RL      R0
P 0036 7B  04      43         jr      c,done
P 0038 E0  E2      44         RR      R2
P 003A 1A  F8      45         djnz    r1,loop
                   46
P 003C 29  F4      47 done:   ld      T0,R2       !load value for detected
                   48                                    bit rate!
                   49 !Delay long enough to clear serial line of bit stream!
P 003E D6  0000*   50         call    DELAY
                   51 !clear receive interrupt request!
P 0041 56  FA  F7  52         and     IRQ,#%F7
                   53
P 0044             54 END     main
                   55 END     bit_rate

   0 ERRORS
ASSEMBLY COMPLETE
```

30 instructions
68 bytes
Execution time is variable based on transmission bit rate.

I/O Functions (Continued)

Port Handshake. Each of Ports 0, 1 and 2 may be programmed to function under input or output handshake control. Table 7 defines the port bits used for the handshaking and the mode bit settings required to select handshaking. To input data under handshake control, the Z8 should read the input port when the $\overline{DAV}$ input goes Low (signifying that data is available from the attached device). To output data under handshake control, the Z8 should write the output port when the RDY input goes Low (signifying that the previously output data has been accepted by the attached device). Interrupt requests IRQ0, IRQ1, and IRQ2 are generated by the falling edge of the handshake signal input to the Z8 for Port 0, Port 1, and Port 2 respectively. Port handshake operations may therefore be processed under interrupt control.

Consider a system that requires communication of eight parallel bits of data under handshake control from the Z8 to a peripheral device and that Port 2 is selected as the output port. The following assembly code illustrates the proper sequence for initializing Port 2 for output handshake.

```
CLR   P2M   !Port 2 mode register: all Port
            2 bits are outputs!
OR    %03,#%40
            !set DAV2: data not available!
LD    P3M,#%20
            !Port 3 mode register: enable
            Port 2 handshake!
LD    %02,DATA
            !output first data byte; DAV2
            will be cleared by the Z8 to
            indicate data available to
            the peripheral device!
```

Note that following the initialization of the output sequence, the software outputs the first data byte without regard to the state of the RDY2 input; the Z8 will automatically hold $\overline{DAV}2$ High until the RDY2 input is High. The peripheral device should force the Z8 RDY2 input line Low after it has latched the data in response to a Low on $\overline{DAV}2$. The Low on RDY2 will cause the Z8 to automatically force $\overline{DAV}2$ High until the next byte is output. Subsequent bytes should be output in response to interrupt request IRQ2 (caused by the High-to-Low transition on RDY2) in either a polled or an enabled interrupt mode.

	Port 0	Port 1	Port 2
Input handshake lines	$P3_2 = \overline{DAV}$ $P3_5 = RDY$	$P3_3 = \overline{DAV}$ $P3_4 = RDY$	$P3_1 = \overline{DAV}$ $P3_6 = RDY$
Output handshake lines	$P3_2 = RDY$ $P3_5 = \overline{DAV}$	$P3_3 = RDY$ $P3_4 = \overline{DAV}$	$P3_1 = RDY$ $P3_6 = \overline{DAV}$
To select input handshake:	set bit 6 & reset bit 7 of P01M (program high nibble as input)	set bit 3 & reset bit 4 of P01M (program byte as input)	set bit 7 of P2M (program high bit as input)
To select output handshake:	reset bits 6, 7 of P01M (program high nibble as output)	reset bits 3, 4 of P01M (program byte as output)	reset bit 7 of P2M (program high bit as output)
To enable handshake:	set bit 5 of Port 3 ($P3_5$); set bit 2 of P3M	set bit 4 of Port 3 ($P3_4$); set bits 3, 4 of P3M	set bit 6 of Port 3 ($P3_6$); set bit 5 of P3M

Table 7. Port Handshake Selection

Arithmetic Routines

This section gives examples of the arithmetic and rotate instructions for use in multiplication, division, conversion, and BCD arithmetic algorithms.

Binary to Hex ASCII. The following module illustrates the use of the ADD and SWAP arithmetic instructions in the conversion of a 16-bit binary number to its hexadecimal ASCII representation. The 16-bit number is viewed as a string of four nibbles and is processed one nibble at a time from left to right, beginning with the high-order nibble of the lower memory address. %30 is added to each nibble if it is in the range 0 to 9; otherwise %37 is added. In this way, %0 is converted to %30, %1 to %31, . . . %A to %41, . . . %F to %46. Figure 5 illustrates the conversion of RR0 (contents = %F2BE) to its hex ASCII equivalent; the destination buffer is pointed to by RR4.

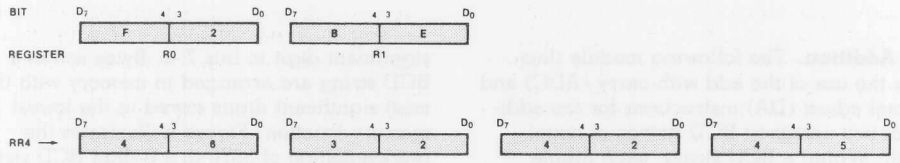


Figure 5. Conversion of (RR0) to Hex ASCII

```
Z8ASM      2.99     INTERNAL RELEASE
LOC     OBJ CODE     STMT SOURCE STATEMENT

                       1 ARITH   MODULE
                       2 GLOBAL
P 0000                 3 BINASC  PROCEDURE
                       4 !*****************************************************
                       5  Purpose =       To convert a 16-bit binary
                       6                   number to Hex ASCII
                       7
                       8  Input =         RR0 = 16-bit binary number.
                       9                   RR4 = pointer to destination
                      10                         buffer in external memory.
                      11
                      12  Output =        Resulting ASCII string (4 bytes)
                      13                  in destination buffer.
                      14                  RR4 incremented by 4 .
                      15                  R0,R2,R6 destroyed.
                      16 *****************************************************!
                      17 ENTRY
                      18
P 0000 6C  04         19          ld      R6,#%04 !nibble count!
P 0002 F0  E0         20 again:   SWAP    R0      !look at next nibble!
P 0004 28  E0         21          ld      R2,R0
P 0006 56  E2  0F     22          and     R2,#%0F !isolate 4 bits!
                      23 !convert to ASCII : R2 + #%30 if R0 in range 0 to 9
                      24                   else  R2 + #%37 (in range 0A to 0F)
                      25 !
P 0009 06  E2  30     26          ADD     R2,#%30
P 000C A6  E2  3A     27          cp      R2,#%3A
P 000F 7B  03         28          jr      ult,skip
P 0011 06  E2  07     29          ADD     R2,#%07
P 0014 92  24         30 skip:    lde     @RR4,R2         !save ASCII in buffer!
P 0016 A0  E4         31          incw    RR4             !point to next
                      32                                   buffer position!
P 0018 A6  E6  03     33          cp      R6,#%03 !time for second byte?!
P 001B EB  02         34          jr      ne,same_byte    !no.!
P 001D 08  E1         35          ld      R0,R1           !2nd byte!
```

```
                        36 same_byte:
P 001F 6A  E1           37          djnz    R6,again
P 0021 AF               38          ret
P 0022                  39 END      BINASC
                        40 END      ARITH

   0 errors
Assembly complete
```

15 instructions
34 bytes
120.5 µs (average)

BCD Addition. The following module illustrates the use of the add with carry (ADC) and decimal adjust (DA) instructions for the addition of two unsigned BCD strings of equal length. Within a BCD string, each nibble represents a decimal digit (0-9). Two such digits are packed per byte with the most significant digit in bits 7-4. Bytes within a BCD string are arranged in memory with the most significant digits stored in the lowest memory location. Figure 6 illustrates the representation of 5970 in a 6-digit BCD string, starting in register %33.

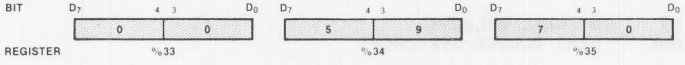


Figure 6. Unsigned BCD Representation

```
Z8ASM     2.0
LOC    OBJ CODE    STMT SOURCE STATEMENT

                     1 ARITH    MODULE
                     2 CONSTANT
                     3  BCD_SRC := R1
                     4  BCD_DST := R0
                     5  BCD_LEN := R2
                     6 GLOBAL
P 0000               7 BCDADD   PROCEDURE
                     8 !*****************************************************
                     9  Purpose =       To add two packed BCD strings of
                    10                   equal length.
                    11                   dst  <-- dst + src
                    12
                    13  Input =         R0 = pointer to dst BCD string.
                    14                   R1 = pointer to src BCD string.
                    15                   R2 = byte count in BCD string
                    16                        (digit count = (R2)*2 ).
                    17
                    18  Output =        BCD string pointed to by R0 is
                    19                   the sum.
                    20                   Carry FLAG = 1 if overflow.
                    21                   R0 , R1 as on entry.
                    22                   R2 = 0
                    23 *****************************************************!
                    24 ENTRY
                    25
P 0000 02  12       26          add      BCD_SRC,BCD_LEN !start at least... !
P 0002 02  02       27          add      BCD_DST,BCD_LEN !significant digits!
P 0004 CF           28          rcf                      !carry = 0!
```

Arithmetic Routines (Continued)

```
                        29 add_again:
P 0005 00   E1          30          dec     BCD_SRC          !point to next two
                        31                                    src digits!
P 0007 00   E0          32          dec     BCD_DST          !point to next two
                        33                                    dst digits!
P 0009 E3   31          34          ld      R3,@BCD_SRC      !get src digits!
P 000B 13   30          35          ADC     R3,@BCD_DST      !add dst digits!
P 000D 40   E3          36          DA      R3               !decimal adjust!
P 000F F3   03          37          ld      @BCD_DST,R3      !move to dst!
P 0011 2A   F2          38          djnz    BCD_LEN,add_again  !loop for next
                        39                                      digits!
P 0013 AF               40          ret                      !all done!
                        41
P 0014                  42 END      BCDADD
                        43 END      ARITH

    0 ERRORS
ASSEMBLY COMPLETE
```

11 instructions
20 bytes
Execution time is a function of the number of bytes (r) in input BCD string:
20 μs + 12.5 (n - 1) μs

Multiply. The following module illustrates an efficient algorithm for the multiplication of two unsigned 8-bit values, resulting in a 16-bit product. The algorithm repetitively shifts the multiplicand right (using RRC), with the low-order bit being shifted out (into the carry flag). If a one is shifted out, the multiplier is added to the high-order byte of the partial product. As the high-order bits of the multiplicand are vacated by the shift, the resulting partial-product bits are rotated in. Thus, the multiplicand and the low byte of the product occupy the same byte, which saves register space, code, and execution time.

```
Z8ASM      2.99     INTERNAL RELEASE
LOC     OBJ CODE     STMT SOURCE STATEMENT

                       1 ARITH   MODULE
                       2 CONSTANT
                       3  MULTIPLIER     :=      R1
                       4  PRODUCT_LO     :=      R3
                       5  PRODUCT_HI     :=      R2
                       6  COUNT          :=      R0
                       7 GLOBAL
P 0000                 8 MULT    PROCEDURE
                       9 !***************************************************
                      10  Purpose =      To perform an 8-bit by 8-bit unsigned
                      11                 binary multiplication.
                      12
                      13  Input =        R1 = multiplier
                      14                 R3 = multiplicand
                      15
                      16  Output =       RR2 = product
                      17                 R0   destroyed
                      18 ***************************************************!
                      19 ENTRY
P 0000 0C   09        20          ld      COUNT,#9         !8 BITS + 1!
P 0002 B0   E2        21          clr     PRODUCT_HI       !INIT HIGH RESULT BYTE!
P 0004 CF             22          RCF                      !CARRY = 0!
P 0005 C0   E2        23 LOOP:    RRC     PRODUCT_HI
P 0007 C0   E3        24          RRC     PRODUCT_LO
P 0009 FB   02        25          jr      NC,NEXT
P 000B 02   21        26          ADD     PRODUCT_HI,MULTIPLIER
P 000D 0A   F6        27 NEXT:    djnz    COUNT,LOOP
```

Arithmetic Routines (Continued)

```
P 000F AF              28          ret
P 0010                 29 END      MULT
                       30 END      ARITH

   0 errors
Assembly complete
```

9 instructions
16 bytes
92.5 µs (average)

Divide. The following module illustrates an efficient algorithm for the division of a 16-bit unsigned value by an 8-bit unsigned value, resulting in an 8-bit unsigned quotient. The algorithm repetitively shifts the dividend left (via RLC). If the high-order bit shifted out is a one or if the resulting high-order dividend byte is greater than or equal to the divisor, the divisor is subtracted from the high byte of the dividend. As the low-order bits of the dividend are vacated by the shift left, the resulting partial-quotient bits are rotated in. Thus, the quotient and the low byte of the dividend occupy the same byte, which saves register space, code, and execution time.

```
Z8ASM     2.0
LOC    OBJ CODE   STMT SOURCE STATEMENT

                    1 ARITH    MODULE
                    2 CONSTANT
                    3  COUNT          :=      R0
                    4  DIVISOR        :=      R1
                    5  DIVIDEND_HI    :=      R2
                    6  DIVIDEND_LO    :=      R3
                    7 GLOBAL
P 0000              8 DIVIDE   PROCEDURE
                    9 !************************************************
                   10  Purpose =       To perform a 16-bit by 8-bit unsigned
                   11                  binary division.
                   12
                   13  Input =         R1 = 8-bit divisor
                   14                  RR2 = 16-bit dividend
                   15
                   16  Output =        R3  = 8-bit quotient
                   17                  R2  = 8-bit remainder
                   18                  Carry flag = 1 if overflow
                   19                             = 0 if no overflow
                   20 ************************************************!
                   21 ENTRY
P 0000 0C  08      22          ld      COUNT,#8        !LOOP COUNTER!
                   23
                   24 !CHECK IF RESULT WILL FIT IN 8 BITS!
P 0002 A2  12      25          cp      DIVISOR,DIVIDEND_HI
P 0004 BB  02      26          jr      UGT,LOOP        !CARRY = 0 (FOR RLC)!
                   27 !WON'T FIT.  OVERFLOW!
P 0006 DF          28          SCF                     !CARRY = 1!
P 0007 AF          29          ret
                   30
                   31 LOOP:     !RESULT WILL FIT.  GO AHEAD WITH DIVISION!
P 0008 10  E3      32          RLC     DIVIDEND_LO     !DIVIDEND * 2!
P 000A 10  E2      33          RLC     DIVIDEND_HI
P 000C 7B  04      34          jr      c,subt
P 000E A2  12      35          cp      DIVISOR,DIVIDEND_HI
P 0010 BB  03      36          jr      UGT,next        !CARRY = 0!
P 0012 22  21      37 subt:    SUB     DIVIDEND_HI,DIVISOR
P 0014 DF          38          SCF                     !TO BE SHIFTED INTO RESULT!
P 0015 0A  F1      39 next:    djnz    COUNT,LOOP      !no flags affected!
```

Arithmetic Routines (Continued)

```
                        40
                        41 !ALL     DONE!
P 0017 10   E3          42          RLC     DIVIDEND_LO
                        43                                  !CARRY = 0: no overflow!
P 0019 AF               44          ret
P 001A                  45 END DIVIDE
                        46 END ARITH

   0 ERRORS
ASSEMBLY COMPLETE
```

15 instructions
26 bytes
124.5 μs (average)

Conclusion

This Application Note has focused on ways in which the Z8 microcomputer can easily yet effectively solve various application problems. In particular, the many sample routines illustrated in this document should aid the reader in using the Z8 to greater advantage. The major features of the Z8 have been described so that the user can continue to expand and explore the Z8's repertoire of uses.

SGS-ATES GROUP OF COMPANIES

Italy - France - Germany - Malta - Malaysia - Singapore - Sweden - United Kingdom - U.S.A.

ORDER CODE : DAP55 0582